The CH3 Design for a Simple Implementation of ADI-3 for MPICH with a TCP-based Implementation

William Gropp and Brian Toonen and who else?

April 10, 2003

Abstract

ADI-3 is a full featured, abstract device interface used in the MPICH implementation of MPI to provide a portability layer that allows access to many performance-oriented features of a wide range of communication systems. ADI-3 is allows research into wide range of implementation issues in MPI. However, because it is full featured, it contains a large number of functions that must be implemented. To both simplify the task of experimenting with MPI implementation issues, a simplified "channel" device is described. This device requires the implementation of only a dozen functions but provides many of the performance advantages of the full ADI-3 interface. This smaller interface, called CH3 (for third version of the channel interface) in turn implements the full ADI-3 interface, providing a simple way to port MPICH to a new platform. To illustrate the implementation issues, an implementation of CH3 using TCP sockets is described.

1 Introduction

This document outlines the CH3 "channel" device implementation of the ADI and sketches an implementation of CH3 on TCP. It defines a specific interface to the low level OS TCP operations, and outlines a way for at least the basic `MPID_` routines to be implemented in terms of these abstract operations. This document is preliminary.

The major goals of this implementation include:

1. Illustrate efficiency by minimizing the overhead on common cases. For example, a send/receive of a single word should generate as few "extra" allocations of internal objects as possible. In particular, this design allows the data to be sent directly without creating a `MPID_Request`. We haven't quite managed that on the receive side.
2. Provide a relatively small interface that can be used to port MPICH to new platforms. This replaces the ADI-2 "channel" interface.
3. Provide an example that can use remote write (put) operations for data transfers.

It is *not* a goal to provide an optimally fast implementation. This is intended to be a relatively simple but reasonably efficient implementation.

2 Outline of the Implementation Structure

The MPID implementation makes use of a number of layers. These layers can be implemented logically (without separate function calls) in order to avoid the overhead of function calls. In the case of the TCP implementation, the cost of function calls is dominated by TCP network overheads.

The MPID routines are implemented in terms of a smaller set of routines that perform relatively simple data communication operations. These are designed so that they can easily be implemented with, for example, TCP, but are not restricted to TCP. For those familar with the "channel device" in ADI-2, these routines represent the ADI-3 version of the channel device interface. These routines are prefixed with `CH3_` (really `MPIDI_CH3_`) to indicate that they belong to this interface; a complete description of the CH3 routines is presented in Appendix A.

Communication: The MPID layer communicates by sending messages consisting of a message header (called a packet header to distinguish it from an MPI message), possibly followed by data. The MPID layer defines (internal to itself, so this discussion only applies to the CH3/TCP implementation of MPID):

Packet Types. An `enum` of types, this describes roughly a dozen kinds of message that are needed to implement MPI message-passing semantics. The packet types and what they contain include:

- **eager_send.** MPI envelope and optional request id; data immediately follows the packet (with no separate header). An MPI envelope contains the data used to match MPI messages: tag, sender's rank, and context id, along with any MPI flow control (for eager messages) and error-checking features such as datatype signatures. The optional request id is sent when message can be cancelled.
- **ready_send.** Same as eager_send but used to perform ready send operations.
- **eager_sync_send.** Same as eager_send but used to perform synchronous send operations. The request id is required.
- **eager_sync_ack** Request id. An acknowledgement that the receiving process has posted a receive matching the send request.
- **rndv_req_to_send.** MPI envelope and send request id.
- **rndv_clr_to_send.** Send and receive request id
- **put.** Address and length, followed immediately by data
- **rndv_send.** send and receive request id, followed immediately by data
- **cancel_send_req.** Send request id to cancel
- **cancel_send_resp.** Send request id and true/false (ack/nak) for was cancelled
- **flow_cntl_update.** Flow control for eager messages and rendezvous requests. This is separate from any low-level flow control, though it may be coordinated with it. For example, we may include knowledge about the size of the socket buffer in this level of flow control.

Additional packet types will be defined to support MPI-2 operations such as RMA.

Packet Format. The actual layout of a packet; for each packet type, there is a corresponding packet format defined by a structure. For version zero, all packet layouts will have the same size in bytes; this simplifies the implementation.

Packet Handlers. The code to be invoked when a packet arives at its destination.

Queues and Connections: (Need some discussion of these, since the requests move between queues and the read/write operations are ordered on a connection.)

CH3 routine brief summary: (I don't think that these are right yet, but we need to start somewhere.) Details in Appendix A.

CH3_Request_create. Create a new request. This is used with `CH3_iSend`, and for cases where a completed request is required (see the discussion of `MPID_Isend`).

CH3_Request_add_ref. Add one to the request's reference count.

CH3_Request_release_ref. Decrement the request's reference count by one. Returns non-zero if the count is non-zero and zero if the count is now zero.

CH3_Request_destroy. Destroy an existing request. This is used after CH3_Request_release_ref returns a count of zero.

CH3_iStartMsg. Begin a message. Return a request if the message has not been completely sent.

CH3_iStartMsgv. Like `CH3_iStartmsg`, but with an `struct iovec`.

CH3_iStartRead. Begin reading data. Return a request if the message has not been completely received.

CH3_iSend. Send data using an existing request.

CH3_iSendv. Like `CH3_iSend`, but with an `struct iovec`.

CH3_iWrite. Like `CH3_iSend`, but the data sent is within the request (the `active_buf` member).

CH3_iRead. Read data to a location specified by a request.

CH3_Progress_xxx. Progress functions. `CH3_Progress` is responsible for dispatching incoming messages.

CH3_Init. Initialize the device and setup the initial communicators.

CH3_Finalize. Finalize the device.

CH3_InitParent. Initialize the parent communicator (if one exists).

CH3_iPut. Nonblocking, contiguous put into remote memory (optional, provided as a hook for non-TCP methods and as a placeholder for the routines necessary for implementing MPI-2 RMA).

The bindings for these routines are:

```
/* Routines that create and destroy requests */
MPID_Request * CH3_iStartMsg( MPID_VC * vc, void * header,
                              MPID_msg_sz_t header_sz )
MPID_Request * CH3_iStartMsgv( MPID_VC * vc, MPID_IOV * iov, int iov_n )
MPID_Request * CH3_Request_create( void );
void MPIDI_CH3_Request_add_ref( MPID_Request * req );
void MPIDI_CH3_Request_release_ref( MPID_Request * req, int * flag );
void CH3_request_destroy( MPID_Request * req );

/* Routines that send or receive data and use an existing request.
   These are used when incrementally processing communication, for
   example, when packing and sending the next segment from a
   non-contiguous datatype.  Each of these is nonblocking and
   indicates completing by decrementing the completion count (cc)
   field in the request. */
void CH3_iSend( MPID_VC * vc, MPID_Request * sreq, void * header,
                MPID_msg_sz_t header_sz )
void CH3_iSendv( MPID_VC * vc, MPID_Request * sreq, MPI_IOV * iov, int iov_n )
void CH3_iWrite( MPID_VC * vc, MPID_Request * sreq )
void CH3_iRead( MPID_VC * vc, MPID_Request * rreq )

/* Routines for progress */
void CH3_Progress_start( void )
void CH3_Progress_end( void )
int CH3_Progress( int is_blocking )
void CH3_Progress_poke( void )
void CH3_Progress_signal_completion( void )

/* Routines for startup/rundown */
int CH3_Init( int * has_args, int * has_env, int has_parent )
void CH3_Finalize( void )
void CH3_InitParent( MPID_Comm * parent )

/* Routines for RMA */
void CH3_iPut( MPID_VC * vc, void * buf, MPID_msg_sz_t buf_sz,
               MPID_RAint offset, MPID_RAint cmpl_flag )
MPID_Request *CH3_iStartRead( MPID_VC * vc, void * buf, MPID_msg_sz_t buf_sz )
```

The use of the letter `i` in the names is meant to emphasize that these are non-blocking in the MPI sense: data is not necessarily transfered before the routine returns and the transfer may continue afterwards (this is what is different between, for example, `CH3_iWrite` and a `write` to a nonblocking socket).

CH3 routines do not handle datatypes; they only handle contiguous data (buf,count) or struct iovec (viewed as bytes). Communication routines are nonblocking, so they must have (a) a connection to which they are attached (so that data is correctly ordered and sent/received) and (b) a request that allows incremental pack/unpack. Each CH3 communication routine, in effect, makes a callback when the communication completes. The action may be as simple as "decrement busy flag and dequeue communication" or as complex as "pack next buffer for sending and send it". However, we do not require the full generality of arbitrary callback routines, so the action to take on completion will be specified by an integer. By chosing this limitation, some operations may be inlined for greater efficiency. The completion actions and associated "upcall" functions supplied by the device will be discussed later.

These routines must be prepared to create actual connections (e.g., establish a socket) if there is no connection already present. It is up to the implementation of the CH3 routines to decide how this is accomplished.

Consequences. We can summarize the requirements for the CH3 interface as:

1. Nonblocking. Correct operation of the code is not dependent on any read or write operation completing when first issued.

2. Contiguous (or simple iovec) data. For simplicity, only contiguous byte ranges (or Unix-style iovec) data moves are handled at the lowest level

3. Correctness. MPI requires that the message envelopes are ordered; data transfers must also arrive in expected order. However, individual data transfers are *not* ordered. Completion of a data transfer guarantees only that all the data have arrived, not the particular order of arrival.

4. Handshakes. Some communication, particularly rendezvous transfers, requires handshakes or cooperation between sender and receiver

5. Performance. Low latency for short messages.

These requirements suggest that there be a single data structure that holds the progress of all communication. In particular, to correctly support the first (nonblocking) requirement, there must be a queue (a queue, not a list or a heap, because of the third (ordering) requirement) of pending data transfer operations. Because of the second (contiguous) requirement, combined with the need to handle general (possibly noncontiguous) MPI datatypes, we must be able to transfer data in parts (segments). This requires having the option of invoking a routine when a data transfer completes (the callback described above); further, it requires that the queue element not be automatically removed from the data transfer queue when the current communication completes because more data for this communication may be on the way.

Because of the fourth (handshake) requirement, once communication has been initiated, the same data structure should be used with any communication that requires a handshake, rather than generating a new data structure for each individual communication operation.

The data structure that satisfies these requirements is the `MPID_Request`. Further, once communication is started (e.g., with `CH3_iSend`), further data transfers are accomplished by either placing the request into the queue of pending data transfers or by updating a request that is already the active request (rather than creating a new request).

To handle the communication of data, the CH3 device needs the following fields in the `MPID_-Request`:

match Matching information: rank, tag, and context.

user_buf. Pointer to the user buffer.

user_count. Number of

datatype. Datatype describing the user buffer.

segment. Segment used to process noncontiguous data.

segment_size. Size of the segment (in bytes).

segment_first. Current offset into the segment.

vc. A pointer to the virtual connection being used to satisfy the request.

iov. I/O vector describing the buffer to be sent or received.

iov_count. Number of entries in the iov.

ca. Completion action. This indicates what operation should be performed when a data transfer is complete.

tmpbuf. Pointer to a temporary buffer

tmpbuf_sz. Size of the temporary buffer (not necessarily the same as the size of the occupying the buffer). The SRBuf flag in the state field indicates if this temporary buffer is a buffer from the send/receiver buffer pool or an unexpected eager

recv_data_sz. Size of the message data.

sender_req_id. Handle of sender's request associated with this request.

state A series of bit fields describing the state of the request.

next A pointer to the next request in the send or receive queue.

Thus, a `ch3` struct is included in the `MPID_Request` structure.

```
typedef struct
{
    MPIDI_Message_match match;
    void * user_buf;
    int user_count;
    MPI_Datatype datatype;
    MPID_Segment segment;
    MPIDI_msg_sz_t segment_size;
    MPIDI_msg_sz_t segment_first;
    MPID_VC *connection;
    MPID_IOV iov;
    int iov_count;
    MPIDI_CA_t  ca;
    void * tmpbuf;
    MPIDI_msg_sz_t tmpbuf_sz;
    MPIDI_msg_sz_t recv_data_sz;
    MPI_Request sender_req_id;
    unsigned state;
    struct MPID_Request * next;
} ch3;
```

The TCP channel also requires fields in the `MPID_Request` to properly handle the communication of data:

iov_offset. Current element in the I/O vector.

pkt. Space for buffering packet headers associated with the request.

Like the CH3 device, the TCP channel adds a structure of its own, namely the `tcp` struct, to the `MPID_Request` structure.

```
typdef struct
{
    int iov_offset;
    MPIDI_CH3_Pkt_t pkt;
} tcp;
```

2.1 Remote Memory Operations in the CH3 Design

This interface includes a remote put operation `CH3_Put`. By having a CH3 routine that can perform a contiguous put to remote memory, we make it easy to experiment with RMA-capable networking (such as VIA or Infiniband) without requiring a completely new device implementation. However, this is an optional routine (at least while we target only MPI-1) and need not be implemented. The packet handler descriptions given show how a put operation can be implemented, even in a TCP/sockets environment.

2.2 Thread Safety

Thread-safety of FOA (currently, inserted request must be marked as unready unless the other receive parameters are passed to FOA so that the request can be atomically created). One possibility is to combine this with the request state and have a state update function that ensures that any modifications are written to memory (e.g., using a write barrier) before another thread might access the request.

2.3 Data Structures

To allow the greatest efficiency, most of the data structures are visible to all layers (below the user-layer). However, parts of the data structures may be defined by and used exclusively by a particular layer. For example, the `MPID_Request` contains both fields used by the MPI implementation layer (e.g., the completion counter) and fields used only by the channel implementation layer. This is essentially a subclassing approach, but implemented directly in C.

The assignment of data structures to layers is as follows.

MPI application layer. (User programs) Owns and allocates `MPI_Status`.

MPI implementation layer. (E.g., implementation of `MPI_Isend` in terms of `MPID_Isend`.) Owns and allocates communicators, datatypes, attributes, groups, files, window objects, keyvals, and error handlers.

MPID Channel device layer. (E.g., implementation of `MPID_Isend` in terms of CH3.) Owns and allocates packets. Defines packet type handlers. Owns and allocates segments.

CH3 implementation layer. Owns and allocates connections and requests.

2.4 Utility Routines

(not written yet) This section should describe the queue operation routines, such as the find-or-post or find-or-allocate (FOA) operations.

CH3U_Request_FDU_or_AEP. Find a request in the unexpected message queue and dequeue it; if one is not found, create a request and add it to the posted receive queue.

CH3U_Request_FDP_or_AEU. Like `CH3U_Request_FDU_or_AEP`, but first checks the posted receive queue, and if not found, adds to the unexpected message queue. This routine is called by a message handler while the previous routine is called by one of the MPI receive routines.

CH3U_Request_FU. Find a matching request in the unexpected queue. Return a pointer to the request or NULL if a matching request was not found. Note: this routine does not remove the request from the unexpected queue.

CH3U_Request_FDU. Given the sender's request handle and matching information, find a matching request in the unexpected queue. Return a pointer to the request or NULL if a matching request was not found.

CH3U_Request_DP. Given a pointer to the request structure, dequeue the request from the posted message queue. Return true if the request was dequeued or false if the request was not found.

CH3U_Request_FDP. Given message match information, find a matching request in the posted queue. If found, dequeue it and return a pointer to the request; otherwise return NULL.

CH3U_Request_create. Initialize the `ch3` structure in a MPID_Request. This routine should be called by `CH3_Request_create`.

CH3U_Request_destroy. Destroy any objects attached to the `ch3` structure in a `MPID_Request`. This routine should be called by `CH3_Request_destroy`.

CH3U_Request_decrement_cc. Atomically decrement the completion counter in the request. If the counter reaches zero, return zero; otherwise return a non-zero value.

CH3U_Request_complete. This is a convenience routine. It calls `CH3U_Request_decrement_cc` followed by `MPID_Request_release` and `CH3_Progress_signal_completion` if all operations associated with the request are complete.

CH3U_Request_load_send_iov. (Re)load the I/O vector in a send request.

CH3U_Request_load_recv_iov. (Re)load the I/O vector in a receive request.

CH3U_Request_unpack_uebuf. Unpack data for a unexpected eager message in the user's message buffer.

CH3U_Request_unpack_srbuf. Unpack data in a send/receive buffer into the user's message buffer.

CH3U_Buffer_copy. Copy the contents of a user's send buffer into a user's receive buffer.

CH3U_SRBuf_alloc. Allocate a send/receive buffer and set req-¿ch3.tmpbuf to point to the buffer. Send/receive buffers are temporary buffers used to hold a portion of the message data when user's buffer is non-contiguous. The desired size of the buffer is specified when calling this routine; however the actual buffer size may be different. `req->ch3.tmpbuf` will contain the actual size of the buffer.

CH3U_SRBuf_free. Free a previously allocated send/receiver buffer.

Bindings:

```
    request = CH3U_Request_FDU_or_AEP( source, tag, context_id, &found )
    request = CH3U_Request_FDP_or_AEU( msg_match, &found )
    request = CH3U_Request_FU( source, tag, context_id )
    request = CH3U_Request_FDU( handle, msg_match )
    flag = CH3U_Request_DP( reqptr )
    request = CH3U_Request_FDP( msg_match )
    CH3U_Request_create( reqptr )
    CH3U_Request_destroy( reqptr )
    CH3U_Request_decrement_cc( reqptr, &count)
    CH3U_Request_complete( reqptr )
    mpi_errno = CH3U_Request_load_send_iov( reqptr, iov, iov_n )
    mpi_errno = CH3U_Request_load_recv_iov( reqptr )
    mpi_errno = CH3U_Request_unpack_uebuf( reqptr )
    mpi_errno = CH3U_Request_unpack_srbuf( reqptr )
    CH3U_Buffer_copy( sbuf, scount, sdatatype, smpi_errno,
                      rbuf, rcount, rdatatype, &data_sz, rmpi_errno )
    CH3U_SRBuf_alloc( reqptr, size )
    CH3U_SRBuf_free( reqptr )
```

2.5 NonContiguous Datatypes

The CH3 interface supports the direct communication of either single contiguous blocks of data or data represented by a standard iovec structure. However, some of the most common noncontiguous datatypes encountered in MPI programs are not efficiently represented by an iovec structure; these include both strided and block-indexed types. While the ADI-3 interface allows the device to directly handle all datatypes, the CH3 interface must pack and unpack datatypes that are noncontiguous. To allow arbitrarily large messages to be sent with the CH3 device, packing and unpacking may be done incrementally. The routines such as `CH3_iWrite` and `CH3_iRead` are used for transfering these incrementally packed buffers.

3 Pseudo-code for some of the `MPID_` routines

This section outlines the pseudocode for some of the major MPID routines. Note that many of these operations only begin a communication. The completion of the communication often takes place with the communication agent. The routines that are implemented within the communication agent, which are called message handlers, are described in Section 4.

3.1 Sending

The code for `MPID_Isend` and `MPID_Send` is shown below. The code for the other MPID send routines is similar, with the appropriate choice of eager (for rsend) or rendezvous (for ssend) operations. The envelopes for ready-send messages should include a flag that indicates that they are ready-send so that the user-error of an unmatched ready-send can be detected and reported.

```
MPID_Isend( )
{
    decide if eager based on message size and flow control
    if (eager) {
        create packet on stack
        fill in as eager send packet
        request = CH3_request_create()
        if (data contiguous)
            CH3_iSendv( request, iov )
        else {
            create pack buffer
            pack into buffer
            CH3_iSendv( request, iov )
            if (complete)
                free pack buffer
            else
                save location of pack buffer in request
            }
        }
    else (rendezvous) {
        create packet on stack
        request = CH3_request_create();
        fill in request
        fill in packet as rndv_req to send (include request id)
        CH3_iSend( request, packet )
    }
    return request
}
```

An alternative to creating packets on the stack is to allow the CH3 layer to provide a way to create a new packet. The semantics would allow simple allocation as above, but would also allow the CH3 layer to provide specially allocated memory. For the near term, however, we will not include this enhancement.

`MPID_Send` is slightly different because we want to avoid allocating a request if possible.

```
MPID_Send( )
{
    decide if eager based on message size and flow control
    if (eager) {
        create packet on stack
        fill in as eager send packet
        if (data contiguous) {
            request = CH3_iStartmsgv( iov )
            // note that the request will be null if the message was sent
            }
        else {
            create pack buffer
            pack into buffer
            request = CH3_iStartmsgv( iov )
            if (!request)
                free pack buffer
            else
                save location of pack buffer in request
          }
    else (rendezvous) {
        .. exactly like MPID_Isend
    }
    return request
}
```

3.2 Receiving

Both `MPID_Irecv` and `MPID_Recv` use similar code.

```
MPID_Irecv( )
```

```
{
    CH3_Progress_poke
    request = MPIDI_CH3U_Request_FPOAU( source, tag, context_id, &found )
    if (found)  {
        /* Message was found in unexpected list.  Eager data is stored
           in the request */
        if (eager) {
            copy data
            free eager buffer used for data
            mark request completed
            }
        else {
            # rendezvous
            create packet on stack
            fill in as rndv_clr_to_send
            CH3_iSend( request, packet )
            }
    }
    else {
        fill in request
        CH3U_Request_change_state( request, waiting for match )
    }
}
```

Note that this code cannot avoid the allocation of a request, even in the case of MPID_Recv and where the data is already available in the socket, since a request must be returned to the user. To optimize for low latency in the case of a small, contiguous transfer, we may want to have a version of MPID_Recv that looks something like

```
    if (datatype is contiguous and small &&
        receive queue for this tag/context/source is empty &&
        no active receive request) {
        Try to read next packet
        if (packet read) {
            if (packet type is eager &&
                MPI envelope matches this receive) {
                transfer data to destination
                if (transfer complete) return # null request since done.
                else {
                    create request, make active
                    return request
                }
            }
            else {
                dispatch packet (e.g., same code as in progress engine)
            }
        }
    }
    /* fall through in case we didn't receive the message */
    MPID_Irecv( ... )
```

An advantage of this is that it avoids both the need for allocating a request and it avoids calling the Progress routine (note that we must still ensure that the progress routine is called sufficiently often). The somewhat complicated tests are necessary to ensure that correct message ordering is preserved. Note that the test "receive queue for this tag etc." need not be perfect in that false negatives (queue may be nonempty) are allowed, since this only drops the code into the `MPID_Irecv` case. For example, a simple test to see if the queue is empty is sufficient.

This example also serves to illustrate why the MPID interface includes blocking receive; there is a potentially important optimization that is otherwise not available without a blocking receive. Also note that this routine cannot be implemented strictly in terms of the `CH3_xxx` routines, because the "try to read next packet" step requires access to the underlying message buffers. Question: we could provide a routine to do this (see the progress engine discussion under "Completion"); should we?

3.3 Completion

The ADI does not provide completion routines that correspond directly to the MPI completion routines (e.g., `MPI_Test`). Instead, there are routines to make progress on communication. To test whether a request is complete, the `busy` flag is checked. The implementation of the `CH3_Progress` routine is shown with the other `CH3` routines in Section A.8.

3.4 Persistent Requests

MPI persistent requests allow the MPI implementation to setup the data structures necessary for communication in advance of initiating the communication. This section briefly sketches these routines and their datastructures.

```
MPID_Send_init()
    act_request = CH3_request_create();
    request     = CH3_request_create();
    request->act_request = act_request;
    request->send_packet = MPIU_Malloc( packet );
    setup fields in request, act_request, send_packet,
        including a free-handler for the request
    return request
```

Rather than use `MPIU_Malloc`, we may want a `CH3` routine that can return a packet that may not be allocated on the stack (e.g., a persistent packet).

The reason that a packet is allocated here rather than off the stack is that this allows us to fill in the packet once during the `MPID_Send_init` step, rather than during each `MPID_Start` step. Note that `MPID_Request_free` must free this packet and the `act_request`.

Persistent requests are initiated by invoking the start function in the request. The implementation of `MPI_Start` and `MPI_Startall` calls the `start_fn` in each request. This provides a common approach for both user-defined requests (called generalized requests in MPI-2) and point-to-point persistent communication requests.

4 Pseudo-code for Message Handlers

These are the routines that are called by the progress engine on receiving a message packet. All of these assume that the entire packet header has been read but that any following data may not yet have been read.

Question: The Unix socket interface supports a "low watermark" setting that guarantees that at least that many bytes are available when `select` or `poll` returns. Should we use this in the code? What are the performance implications?

4.1 EagerSend

Action invoked by the receiver of an eagerly sent message. The data for the message is immediately behind the message header (eagersend packet).

```
    request = MPIDI_CH3U_Request_FPOAU( &packet->msg_match, &found )
    if (found) {
        /* Message was already posted */
        CH3_iRead( request )
    }
    else {
        /* Message is unexpected */
#       ifdef HAVE_ERROR_CHECKING
        if (message is readysend) {
             signal error (and return a message to sender)
             arrange to read and discard data
             (simply allow the read as below; set the state to discard
             when the transfer is complete)
        }
#       endif
        request->active_buf = CH3U_AllocateStorageFromEagerBuffer( len )
        request->active_buf_len = len
        CH3_iRead( request )
    }
```

This routine uses `CH3_iRead` to read the data that follows the message header into the location previously saved in the request (when the request was posted). We don't pass the buffer location to `CH3_iRead` because (a) the location is already present in the request and (b) passing it as an argument to the routine unnecessarily adds to function call overhead.

4.2 RndvReqToSend

Action invoked by the reciever of a rendezvous message.

```
    request = MPIDI_CH3U_Request_FPOAU( &packet->msg_match, &found )
    if (found) {
        if (dest buffer is contiguous)
            create rndv-ok-to-put packet on stack
            fill in packet
            CH3_iSend( request, packet )
        else
            create rndv-ok-to-send packet on stack
            CH3_iSend( request, packet )
    }
#ifdef HAVE_ERROR_CHECKING
    else if (ready-send) {
        return an error message to sender (error msg packet)
        tmp_request = CH3_iStartMsg( packet );
        if (tmp_request) tmp_request->ref_count--;
        MPIDI_CH3U_DeQueue_unexp( request )
    }
#endif
```

Question: it may be possible to reuse the incoming packet as the outgoing packet, thus saving some stores to the packet structure.

4.3 RndvClrToSend

Action invoked by the sender of a rendezvous message on receipt of an acknowledgement from the receiver.

```
Convert request id (in packet) to pointer to request structure
create RndvData packet on stack
iov[0].ptr = address of packet
iov[0].len = sizeof(packet)
if (data contig) {
    iov[1].ptr = data_address
    iov[1].len = data_len
    CH3_iSendv( request, iov, 2 )
}
else {
    Create pack buffer
    Pack first segment worth
    iov[1].ptr = address of pack buffer
    iov[1].len = lenght of packed data
    request->state = segment_sending
    CH3_iSendv( request, iov, 2 )
}
```

4.4 Put

Action invoked by reciever of a put packet.

```
Read Address from packet
request = CH3_iStartRead( address, count )
if (request) {
    Save flag address in request
    set request state so that on completion of data transfer, flag is
        decremented
}
else {
    decrement flag (address provided by packet)
}
```

Note: We do need a request for the put operation to handle incomplete data transfers; by setting the request's reference count, we can ensure that the request is recovered once the data transfer completes. However, in the case where a put is used to provide the data for a rendezvous receive, there is already an available request. Question: do we want a form of put that takes advantage of having an existing request? In that case, instead of the remote flag address, the remote request id can be used.

Note that if the read completes, the returned request is null.

4.5 RndvData

Action invoked by the receiver of data sent in response to an ok to send after a rendezvous message.

```
Convert request id (in packet) to pointer to request structure
/* memory location already set as active buf in request */
CH3_iRead( request )
```

4.6 CancelSend

Action invoked by the receiver of a request to cancel a previously sent RndvReqToSend.

```
Convert request id (in packet) to pointer to request structure
If (request is in unexpected message queue)
    if (already matched) {
        create CancelSendAck(failed) on stack
```

```
            newrequest = CH3_iStartmsg( packet )
        }
        else {
            Remove and discard request
            create CancelSendAck(succeeded) on stack
            newrequest = CH3_iStartmsg( packet )
        }
    else {
        create CancelSendAck(failed) on stack
        newrequest = CH3_iStartmsg( packet )
    }
    if (newrequest) newrequest->ref_count--;
```

Note that the above code must access the message queues atomically in the multi-threaded case in order to preserve correctness. Question: should the queue access part of this be an CH3U utility routine?

4.7 CancelSendAck

Action invoked by the receiver of a request that acknowleges a CancelSend request.

```
    Convert request id (in packet) to pointer to request structure
    Set request to indicate whether cancel succeeded
    If succeeded, remove from pending send list
```

As for the CancelSend message, this must act atomically on the message queue.

4.8 FlowControlUpdate

Flow control is used at the MPI level to control the use of eager buffers and requests for unexpected messages. Possible choices for flow control include IMPI-style control (`http://impi.nist.gov/impi-report/impi-report-node54.html`) or an integrated count of the number of envelopes and buffer space used (IMPI only counts "packets"). Flow control is *not* optional, though the early implementation can ignore this.

One possibility is to include flow control updates on all packets; this ensures that in typical "balanced" communication, a separate flow control packet is never needed.

The original MPICH-1 flow control counted envelopes and data separately, encoding the number of each consumed since the last communication within a single 32-bit integer field in each message packet.

(need to add more details on flow control.) The biggest issue is the handling of eager buffer space. The problem is fragmentation; you can't simply count the number of bytes. One possibility is to allocate space in a buddy system and count the number allocated in each buddy pool (each pool contains identically sized blocks). Note that because the maximum size of an eager message is limited, we don't really need to worry about a message being too large for a single fragment in the eager buffer bool.

5 Message Queues

This section describes the routines for handling the message queues. For thread safety, the queues of posted receive requests and of unexpected messages are accessed atomically rather than through separate routines.

In addition, the match condition is based on data that is passed in the packet and is stored in the type `MPID_Message_match`:

```
typedef struct {
    int32_t tag;
    int16_t source;
    int16_t context_id;
} MPID_Message_match;
```

To exploit longer-word instructions where available, this is really a union:

```
typedef union {
    struct {
    int32_t tag;
    int16_t source;
    int16_t context_id;
    } match;
    int32_t match32[2];
#ifdef HAVE_INT64_T
    int64_t match64;
#endif
} MPID_Message_match;
```

On systems with `int64_t`, the match test is a single line of C code; if the hardware has 64-bit integer operations, it is a single compare instruction. Otherwise, the code shown here must be expanded to test two 32-bit fields.

```
MPIDI_CH3U_Request_FPOAU( int source, int tag, int context_id, int *found )
   /* Look in unexpected queue for a match */
   MPID_Message_match m.match = {tag, source, context_id};
   if (tag != MPI_ANY_TAG && source != MPI_ANY_SOURCE) {
       for (r = unexpected_queue->head; r; r = r->next ) {
           if (r->match.match64 == m.match64) {
               remove r from unexpected queue
               found = 1
               return r
           }
       }
   }
   else {
       /* Must do any match */
       MPID_Message_match mask.match = {0xffffffff, 0xffff, 0xffff};
       if (tag == MPI_ANY_TAG) { mask.match.tag = 0; m.match.tag = 0; }
       if (source == MPI_ANY_SOURCE) {
           mask.match.source = 0; m.match.source = 0; }
       for (r = unexpected_queue->head; r; r = r->next ) {
           if ((r->match.match64 & mask.match64) == m.match64) {
               remove r from unexpected queue
               found = 1
               return r
           }
       }
   }
   found = 0;
   /* If we get here, the message was not found */
   request = CH3_request_create()
   request->match.match = { tag, source, context_id };
   request->mask.match  = { like the above };
   /* Add to tail of posted recieves */
   posted_receive->tail->next = request;
   posted_receive->tail       = request;
   return request
```

The above code shows an example of optimizing for code that does not use wildcards for the tag or source.

The next routine is always called in response to receiving a message packet, and hence simply passes the part of the packet that contains the tag, context, and source values directly to the routine. It is essentially the same as `MPIDI_CH3U_Request_FPOAU`, except that it checks first in the posted receive queue and, if the data is not found, adds to the unexpected receive queue. The wildcard match testing is slightly different as well.

```
MPIDI_CH3U_Request_FUOAP( MPID_Message_match *, int *found )
   /* Look in posted queue for a match */
   for (r = posted_queue->head; r; r = r->next ) {
       if (r->match == (m & r->mask)) {
           remove r from posted queue
           found = 1
           return r
       }
   }
   found = 0;
   /* If we get here, the message was not found */
   request = CH3_request_create()
   request->match = { tag, source, context_id };
   /* Add to tail of unexpected recieves */
   unexp_receive->tail->next = request;
   unexp_receive->tail       = request;
   return request
```

Cancel operations require that requests be removed for the queues.

This routine is required for cancelling sends.

```
MPIDI_CH3U_DeQueue_unexp( int handle )
    for (r = unexpected->head; r; r = r->next) {
        if (r->handle == handle) {
             remove r from list and return it
        }
    return null
```

(Recall that the integer id of the request is called the `handle`.)

This routine is required for cancelling receives

```
MPIDI_CH3U_DeQueue_posted( int handle )
    for (r = posted->head; r; r = r->next) {
        if (r->handle == handle) {
             remove r from list and return it
        }
    return null
```

The above are appropriate for rare operations such as cancel that can afford to search through potentially long lists and can even use locks to guarantee exclusive access to the data structures. Note that the description of the cancel routines do not use these routines but they should.

5.1 Debugger Interface

(still to do; this section will describe how to support the message queue operations that are part of the debugger interface. Note that the debugger interface provides for both a receive and send queue interface; the debugger can get a list of the pending send and receive operations, along with the unexpected messages.

6 Implementing `mpiexec`

(This is a temporary spot for these remarks, since they may apply to more than just the TCP device.)

There are multiple possible implementations for `mpiexec` and each has its own advantages and disadvantages. We might implement all of them, but we should implement the quickest-to-implement first.

As MPD console program. This is ready-to-go as a minor change to `mpdcon.c`, except that the BNR interface might need to be updated to match the current specification. A version of `MPI_Info` is needed for the new `BNR_Spawn`, but not for anything else, so MPI-1 routines should be OK. I.e, the database part of BNR is already running. All handling of `stdio` is done.

As a BNR program. This requires the above plus implementation of `BNR_Spawn`, at least for use by console. It could also be built to interact with a scheduler. This (using `BNR_Spawn` to start the initial processes as well as for the implementation of `MPI_Spawn`) was the "original" plan.

As an MPI program. This is the idea in the current MPICH2 document. It relies on `MPI_Connect`, etc. It requires the above plus the MM component of the BNR interface, to set up the connections. This approach as the advantage of providinga "universal" `mpiexec`.

As an "immediate scheduler". This makes `mpiexec` into a stand-in for the scheduler component of the Scalable System Software Project. It is much like the "MPD console" option, but instead of using the existing console code to contact a local MPD, it sends the standard XML defined by the SSS project to the MPD, which is standing in for an arbitrary process startup component. It requires hooking in an XML parser like `xpat` into the MPD and having `mpiexec` emit XML code.

As a "one-host-only" process starter. The `mpiexec` process could simply fork the application processes. This requires a new but simple implementation of the put/get/fence part of the BNR interface. The original `mpiexec` process could become the database server part after forking.

The first and last options seem to present the shortest paths to getting something running that we can use to debug the coming avalanche of code.

Any of these need to contain the argument-processing code for the defined standard arguments to `mpiexec`. These are defined in Volume 1 of *MPI—The Complete Reference*, starting on page 353. There are multiple approaches to dealing with arguments.

Plain. Use straightforward code as in `p4_args.c`.

Fancy. Use an "options database" approach, as in PETSc.

We will want to do both, but the first option can be implemented immediately, especially if we postpone some of the more elaborate argument lists and require that those be used with a file. We have to define the format of the file for use with the `-file` option. There are three possibilities.

Keyword=value pairs. This is easy to read, and we can use the parsing routines from MPD, so we are practically already done.

XML. We could match the process-startup file to the format of a process-startup request as being defined by the Scalable Systems Software Project. This would be sort of cool. Validating XML parsers in C exist.

Custom Format. We could define our own formats, so that we could express anything whatsoever. We could use multiple formats to match other software that we might find it useful to be compatible with, such as schedulers and other process managers.

Again, we might want to implement all three of these, since each has advantages. The quickest option is the first. However, we could easily implement an XML-style version of keyword/value pairs using a format such as

```
<MPICH keyword=value />
```

and have both the first and second choices at the same time.

A CH3 Routines and Data Structures

This section provides pseudocode for a TCP implementation of the CH3 routines.

General note: in if-else code, the most likely case should be placed first. This is both faster and moves the most common code branch to the top, where it makes it easier to grasp the intent of the code.

A.1 Data Structures

There are two primary data structures: one for virtual connections and one for the device as a whole...*needs revision...*

Process Specific Information (CH3 Device).

```
typedef struct {
    MPID_Request * recv_posted_head;      /* List of posted receives */
    MPID_Request * recv_posted_tail;
    MPID_Request * recv_unexpected_head;  /* List of unexpected receives */
    MPID_Request * recv_unexpected_tail;
} MPIDI_Process_t;
```

The receive lists are held on the device to simplify handling of wildcard receives.

Process Specific Information (TCP Channel).

```
typedef struct {
    MPIDI_CH3I_Progress_group_t * pg;     /* My process group */
} MPIDI_CH3I_Process_t;
```

Process groups.

```
typedef struct {
    volatile int ref_count;
    char * kvs_name;                      /* name of the PMI keyval space */
    int size;                             /* number of processes */
    struct MPIDI_VC * vc_table;           /* table of virtual connections
                                             (one per process) */
} MPIDI_CH3I_Process_group_t;
```

Connections.

```
typedef struct {
    int ref_count;                        /* number of comunicators using this
                                             connection */
    int lpid;                             /* local process ID for the partner
                                             of this connection (used to
                                             implement group routines)*/
    MPIDI_CH3_VC_DECL                     /* channel fields */
} MPIDI_VC;
```

```
#define MPIDI_CH3_VC_DECL
typedef struct {
    MPIDI_CH3I_Process_group_t pg;        /* process group containing remote
                                             process */
    int pg_rank;                          /* rank in the process group */
    struct MPID_Request * sending_head,   /* Queue of pending sends */
    struct MPID_Request * sending_tail;
    MPIDI_CH3I_VC_state_t state;          /* TCP connection state */
    int poll_elem;                        /* element in the poll array */
    int fd;                               /* fd for the socket (cached) */
} tcp;
```

The pending sends have a head and a tail pointer because it is a queue and we want the operations of insert and delete on this queue to be fast.

A.2 Requests

`CH3_Request_create` returns a new request allocated from the request pool. The initial implementation uses the `MPIU_Handle` module to manage this pool. As a result, the `id` field in the request structure automatically contains an integer handle for the request object that may be used as the `MPI_Request` handle.

```
CH3_Request_create()
{
    request = MPIU_Handle_obj_alloc(&MPID_Request_mem);
    request->ref_count = 1;
    return request;
}
```

This routine requires that `MPIU_Handle_obj_alloc` be thread safe in a multithreaded environment, preferably without using locks.

Question: What should this routine do if it cannot allocate a new request? Is that a fatal error? What about the fault-tolerant case? Should the code look something like

```
CH3_request_create()
{
    request = MPIU_Handle_obj_alloc(&MPID_Request_mem);
    if (!request) {
       int limit = 200;
       do {
          CH3_Progress( 0 );
          request = MPIU_Handle_obj_alloc(&MPID_Request_mem);
       }
       while (!request && limit--) {
       if (!request) {
           MPID_Abort( ) // Give up
    }
    request->ref_count = 1;
    return request;
}
```

`CH3_Request_destroy` releases the resources used by an existing request back to the request pool.

```
CH3_Request_destroy( MPID_Request * request )
{
    request->ref_count -= 1;
    if (request->ref_count == 0)
    {
        MPIU_Handle_obj_free(&MPID_Request_mem, request);
    }
}
```

This routine requires that `MPIU_Handle_obj_free` be thread safe in a multithreaded environment, preferably without using locks.

A.3 iStartMsg

`CH3_iStartMsg` is called to send a message. It returns a request if the message is not completely sent.

```
MPID_Request *CH3_iStartMsg( MPID_VC *, void *header, int header_count )
   if there is no active request on this VC
      write the header.
      if the entire header is written, return null
      else /* see note below */
          create a request
          fill it in with the remaining data to write
          make this the active request
          return request
   else
      create a request
      fill it in with the remaining data to write
      add to the pending send queue
      return request
```

One possible variation is to eliminate the “else” branch labeled “see note below” and let the code fall through into else branch that creates and inserts the request into the pending send queue.

```
CH3_iStartMsgv( MPID_VC *, struct iovec *iov, int count )
Like CH3_iStartMsg, but for all of the data in the iov
```

A.4 iStartRead

This is used only in the handler for a put packet type. It reads data and generates a new request only if not all of the requested data was read. It must be implemented *only* if `CH3_Put` is implemented.

```
CH3_iStartRead( MPID_VC *, void *buf, int count )
   Try to read count bytes to buf
   if (all read) return 0;
   /* otherwise */
   create a new request
   request->active_buf = (char *)buf + bytes read
   request->count_left = count - bytes read
   return request
```

A.5 iSend

iSend it called to send data using an existing request. However, the data to send is passed in separately (it is not part of the request).

```
CH3_iSend( MPID_VC *, MPID_Request *, void *buf, void *count )
    if (!vc->sending_head) { /* no active send requests */
        try to send data.
        if (all sent) {
            switch (request->comm_state)
                0: request->cc--;
                   if (request->ref_count == 0) recover request
               >0: (request->update)(request) /* effectively "invoke
                    MPIDI_CH3_Request_update( request )" */
         }
    else {
        make active request
        }
    }
    else {
        request->active_buf = buf; request->count_left = count;
        add request to top of pending send queue
    }
```

The member cc of the request is the "completion counter".

Question: Can we combine the reference count and the completion count, so that only a single flag must be tested?

CH3_iSendv is like CH3_iSend, except a struct iovec is used instead of a buf,count pair. The special action on a comm_state of zero is used to avoid the overhead of a function call in the common case (for relatively short, contiguous messages) that no further work needs to be done.

A.6 iRead

iRead is used to transfer data into an already created request.

```
CH3_iRead( MPID_VC *, MPID_Request *request )
    read upto request->count_left from fd associated with this VC
        to request->active_buf
    if (all read) {
        switch (request->state)
            0: decrement request->cc
               remove from active list
               if (request->ref_count == 0) place request on avail list
            >0: invoke MPIDI_CH3_request_update( request )
    }
```

A.7 iWrite

iWrite is called to continue a data transfer. This is used by the segment processing code to handle the incremental packing and sending of noncontiguous datatypes.

```
CH3_iWrite( MPID_VC *, MPID_Request * )
    if request is not the active request on this VC, internal error
    try to write on fd associated with MPID_VC.
    record amount of data written in request.
    If all data written
        switch(request->state)
            0: decrement request->cc and remove this request from
               the active list.  If ref_count is 0, return request to avail
            >0: invoke MPIDI_CH3_Request_update( request )
    return
```

The request update routine handles incremental packing; typically, it uses MPIR_Segment_pack to pack the next segment of bytes and then calls CH3_iWrite to continue writing the data (note that the request is left on the active list in case some other thread want to start sending data on this connection).

IMPLEMENTATION NOTE: The final case may result in deep recursion for large messages. This problem must be solved eventually. For now, we simply verify that the call stack does not get too deep and hope for the best.

A.8 Progress

The progress routine may be implemented on single-threaded systems as

```
CH3_Progress( int isblocking )
{
    select/poll on all fd's (wait is blocking, test is not)
    For each fd, find associated MPID_VC. (perhaps using fd_to_vc[] array)
    if (write) {
        // fd was set because data is waiting
        do {
            send data described by active request in MPID_VC.
            If (all data sent)
                invoke request_state_method.
                if no active send requests, clear need-to-write
            } while (can write and pending writes)
    }
    if (read) {
        do {
            switch on state
                case reading pkt hdr:
                    switch on packet headr
                        call fcn associated with pkt type
                        (These are MPID functions, not TCP functions,
                         since pkts are in MPID layer.  Call them
                         MPIDI_CH3U_xxx)
                    end switch
                case reading data to known address:
                     read more data.
                     if (complete)
                         invoke request_state_method
                case establishing new connection:
                     ... // this is the hard part, of course
            end switch
        } while (can read)
    }
}
```

To make this more efficient, maintain the data structures needed with the `poll` or `select` calls with the active fd's: the ones on which there are either pending writes or on which connections for reading have been established. Such a structure might include

```
    typedef struct {
       ... whatever poll needs ...
       int writes_pending;
       ...
    } MPIDI_CH3_Progress_data_t;
```

The above version of `CH3_Progress` is for single-threaded implementations. Multi-threaded versions must keep track of whether any completions occurred after `CH3_Progress_start` was called.

An alternative to having the CH3 layer contain the progress engine is to define some additional read and write routines at the CH3 level and then implement the progress routine in terms of those calls. If there were such routines, they could also be used to implement MPID_Recv.

The routines CH3_Progress_start and CH3_Progress_end can be no-ops in the single-threaded version. The routine CH3_Progress_poke can be CH3_Progress(0) (i.e., nonblocking call to progress), and can be defined as a macro to avoid the function call overhead.

On multi-threaded systems, the implementation of the progress routines may use any of the usual techniques, including condition variables.

A.9 Startup and Rundown

```
CH3_Init( int *has_args, int *has_env, int *has_parent )
```

This routine must

- Make any calls to BNR
- Set the size, rank, and MPID_VC fields in comm_world and comm_self
- Set the return values appropriately. has_parent should be set if there a parent communicator that must be initialized.

```
CH3_Finalize(void)
```

This routine should free any memory and other resources.

```
CH3_InitParent( MPID_Comm *parent )
```

Initialize the size, rank, and MPID_VC fields in the communicator comm_parent. CH3_InitParent is called only if CH3_Init returned true in has_parent. This allows the implementation of MPI_Init to create the parent communicator only if it is needed, and then initialize it in a separate step.

A.10 RMA

```
CH3_iPut( MPID_VC *, void *buf, int count, MPID_RAint offset,
          MPID_RAint cmpl_flag )
```

The remote addresses (MPID_RAint) cannot be of type MPI_Aint because MPI_Aint is the size of an address on the calling system, not necessarily on the target system.

B Other Comments

One optimization that is mentioned in the MPICH-2 coding document [?] in the discussion of MPI_-Sendrecv may be valuable to the TCP implementation. This involves piggy-backing rendezvous acknowledgements onto data transfers where possible. For the case when sendrecv is used to exchange data, this can reduce the number of separate communication steps. The design given here provides no way to make this optimization. Question: do we want to allow an extension? What would it cost other cases? Can we do it from within MPI_Sendrecv and/or MPI_Waitall in such a way that we don't penalize other communication patterns? What is the potential benefit? Should we just concentrate on RMA instead?